INF

Studiengang
Medien- und
Kommunikationsinformatik

Hochschule Reutlingen
Reutlingen University

Uwe Kloos, Natividad Martínez, Gabriela Tullius (Hrsg.)

Informatics Inside:

Human-Centered Computing

Informatik-Konferenz an der Hochschule Reutlingen
30. April 2014

ISBN 978-3-00-045427-1

Impressum

Anschrift:
Hochschule Reutlingen
Reutlingen University
Fakultät Informatik
Medien- und Kommunikationsinformatik
Alteburgstraße 150
D-72762 Reutlingen

Telefon: +49 7121 / 271-4002
Telefax: +49 7121 / 271-4042

E-Mail: infoinside@reutlingen-university.de
Internet: http://www.infoinside.reutlingen-university.de

Hochschule Reutlingen
Reutlingen University

Herstellung und Verlag: Hochschule Reutlingen
ISBN 978-3-00-045427-1

Inhaltsverzeichnis

Gestenerkennung & Augumented Virtuality

Softwaretechnik

Entwicklung Mobiler Anwendungen

Virtuelle Welten

Umgebungswahrnehmung von agentenbasierten simulierten Menschmodellen in virtuellen Welten im Kontext C3D*

André Antakli
Reutlingen University
Andre.Antakli@Student.Reutlingen-University.DE

Abstract

Das Ziel dieser Arbeit war die Umsetzung eines Wahrnehmungsensors für Softwareagenten, die über ein virtuelles Menschmodell in einer dreidimensionalen Umgebung agieren. Hierbei sollen die Agenten über den Sensor in der Lage sein, semantische Informationen zu geometrischen Objekten in der Umgebung zu erhalten. Hierfür wurden zwei Verfahren umgesetzt, die das menschliche Sehen simulieren, indem Objekten erkannt werden wenn diese innerhalb eines Sichtfeld liegen. Ein Problem das dabei gelöst werden muss, ist die Identifizierung möglicher Verdeckungen der Objekte. Ein Ansatz dieses Problem zu lösen, ist der Ray-Tracing Ansatz welcher für das erste Verfahren umgesetzt wurde. Das zweite Verfahren verwendet den Occlusion-Culling Ansatz. Auswertungen beider Verfahren haben gezeigt, dass der Ray-Tracing Ansatz eine schnellere Laufzeit aufweist, der Occlusion-Clulling Ansatz jedoch mehr unverdeckte Objekte im Sichtfeld erkennt.

Schlüsselwörter

Softwareagent, Wahrnehmung, Wahrnehmungssensor, virtuelle Welt, Sichtbarkeitsproblem

CR-Kategorien

I.2.10 [**Vision and Scene Understanding**]: Perceptual reasoning

1 Einleitung

Die vorliegende Arbeit wurde im Rahmen des Projektes Collaborate3D (C3D) am deutschen Forschungszentrum für künstliche Intelligenz (DFKI) in Saarbrücken durchgeführt. Ziel dieses Projektes ist die Entwicklung eines Systems, in dem die kollaborative Bearbeitung von Problemstellungen in einer immersiven dreidimensionalen Umgebung möglich ist. Diese Umgebung wird mit XML3D[1] umgesetzt und über den Webbrowser dargestellt. Um ein Szenario simulieren zu können, werden in C3D intelligente Systeme verwendet um beispielsweise menschliches Verhalten nachzuahmen. [19] Ein Szenario welches in C3D betrachtet wird und zentraler Bestandteil dieser Arbeit ist, ist die Wegfindungssimulation eines Einkäufers in einem Supermarkt. Der simulierte Einkäufer erhält dafür eine Art Einkaufsliste mit Objekten die dieser selbstständig in der 3D-Welt ausfindig machen muss. Dabei findet die Wegfindung durch zuvor erlerntem Wissen und der Auswertung der Umgebung statt. Der Einkäufer wird mit Hilfe

*

Betreuer Hochschule: Prof. Dr. Uwe Kloos
Hochschule Reutlingen
Uwe.Kloos@Reutlingen-University.de
Betreuer Firma: Ingo Zinnikus
DFKI GmbH
Ingo.Zinnikus@DFKI.DE

Informatics Inside 2014
Wissenschaftliche Vertiefungskonferenz 2014
30. April 2014, Hochschule Reutlingen

[1] XML3D: http://xml3d.org

eines Softwareagenten, der in der dreidimensionalen Umgebung als virtuelles Menschmodell repräsentiert wird, simuliert. Zum Thema Softwareagent, siehe [7, 5].
Ziel der vorliegenden Arbeit war es, einem Softwareagenten im Zuge von C3D und im speziellen für das Supermarktszenario, die Möglichkeit zu geben, die dreidimensionale Umgebung wahrzunehmen, damit dieser anhand dieser Informationen Schlussfolgerungen ziehen und Aktionen planen kann. Für die Wahrnehmung der dreidimensionalen Welt wurden zwei Verfahren umgesetzt, die in diesem Paper erläutert werden. Nachfolgend wird von einem virtuellen Agenten gesprochen, wenn ein Softwareagent in einer dreidimensionalen Umgebung als virtuelles Menschmodell repräsentiert wird und mit diesem in dieser Umwelt agiert.
Im nachfolgenden Abschnitt werden drei verschiedene Methoden aus der Literatur vorgestellt, wie die Wahrnehmung eines virtuellen Agenten in einer dreidimensionalen Welt simuliert werden kann. In den Abschnitten 3 und 4 werden die für C3D umgesetzten Methoden vorgestellt und miteinander verglichen. Anschließend werden in Abschnitt 5 die Ergebnisse zusammengefasst und ein Ausblick auf die zukünftige Arbeit gegeben.

2 State of the Art

In [12, 6, 11] wird meist unter zwei Wahrnehmungsmethoden unterschieden: künstliche und synthetische Wahrnehmung. Wird eine reale Umgebung durch ein Computersystem wahrgenommen, spricht man von einer künstlichen Wahrnehmung. Diese wird meist in Robotersysteme eingesetzt, die ihre Umwelt über Sensoren auswerten müssen um in ihr agieren zu können. Hierdurch erhält das Robotersystem die semantischen Informationen und somit die Bedeutung von Objekten in dessen Umwelt, im Falle eines eingesetzten Kamerasystems, nur über die Auswertung der Bildinformationen. Die synthetische Wahrnehmung hingegen, ist die simulierte Wahrnehmung in einer virtuellen dreidimensionalen Welt. [6, 11] Die semantischen Informationen der virtuellen Welt können dabei dem virtuellen Agenten zum Beispiel über den Zugriff auf Datenbankinformationen bereitgestellt werden. [15] Im Gegensatz zu einem Robotersystem kann ein virtueller Agent somit Zugriff auf alle Informationen seiner virtuellen Umwelt haben. Nachfolgend werden drei Methoden vorgestellt, die bei der Simulation der Wahrnehmung von Agenten in virtuellen Welten eingesetzt werden.

2.1 Filterbasierte Methoden

Eine Methode wie ein virtueller Agent seine Umgebung wahrnimmt, ist die Filterung von Informationen beispielsweise anhand seiner Position in der virtuellen Welt. Die semantische Bedeutung der virtuellen Welt und deren interne Objekt-Beziehungen werden zum Beispiel in einer Datenbank gespeichert und je nach Zustand des Agenten für diesen freigegeben. [4, 2] In [4] wird die dreidimensionale Welt aus zwei Objekttypen aufgebaut: geometrische Objekte die visuell in der Szene vertreten sind und Objekte die in der Szene vorhanden aber nicht visuell repräsentiert werden. Die virtuelle Umgebung wird in Bereiche unterteilt die in semantischer Relation zueinander stehen. Diese Bereiche werden als nicht sichtbare Objekte in der Szene repräsentiert. Der Agent erhält hierdurch die semantischen Informationen eines Bereichs, über die Kennung des Objektes in der sich dessen virtuelles Menschmodell befindet. In [2] besitzt ein virtueller Agent mehrere Filter um seine Umwelt wahrzunehmen. So besitzt dieser beispielsweise einen Sichtfilter. Über diesen werden nur semantische Informationen von dreidimensionalen Objekte an den Agenten freigegeben, die sich in einem simulierten Sichtfeld befinden. Zudem werden Filter eingesetzt die nur für kontextspezifische Objekte gelten. Wie in [4] wird auch in [2] die virtuelle Welt hierarchisch in Bereiche unterteilt.

2.2 Falschfarben-Methode

Ein Vorgehen, bei dem das „Erkennen“ von Objekten simuliert wird, ist die sogenannte Falschfarben-Methode (eng.: False coloring) Dieses Verfahren wird zum Beispiel in [12, 6, 8, 10] eingesetzt. Bei dieser Methode lokalisiert der virtuelle Agent über ein

für ihn generiertes 2D-Bild Objekte in seiner Umgebung. Das Bild wird dabei für jeden virtuellen Agenten, abhängig von dessen Position und Ausrichtung gerendert. Die Objekte im gerenderten Bild besitzen dabei einen eindeutigen Farbwert, siehe Abbildung 1. Der Farbwert wird hierfür anhand der Objekt-Identifikationsnummer generiert. Wird bei der Bildanalyse ein Farbwert ermittelt, kann dieser wieder in eine Objekt-Identifikationsnummer übersetzt werden. Mit dieser Identifikationsnummer können im Anschluss die semantischen Objektinformationen aus zum Beispiel der Datenbank des Simulationssystems ausgelesen werden. Wird das Bild ausgewertet, erhält das Wahrnehmungsmodul des Softwareagenten Informationen über „sichtbare“ Objekte durch die in den Pixeln gespeicherten Farbwerte. Dieses Verfahren ähnelt somit dem künstlichen Wahrnehmung. Bei dem Renderingprozess werden Texturen oder andere Effekte wie Schattenwurf nicht beachtet. [15, 6, 7, 11]

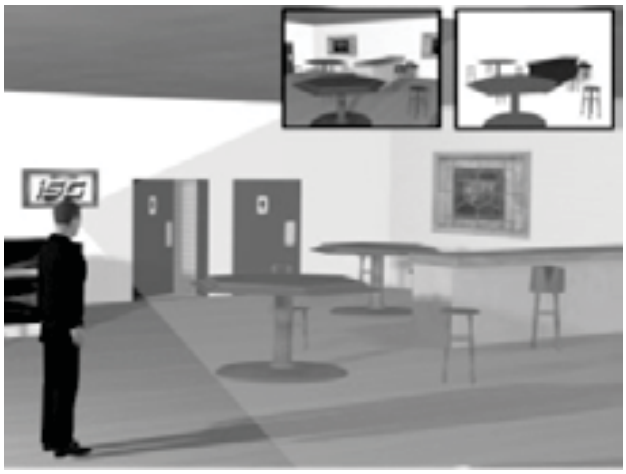

Abbildung 1: Falschfarbenmethode (oben links: Sichtfeld virtueller Agent; oben rechts: Falschfarbenversion) [12]

2.3 Ray-Tracing Methode

Die Raytracing-Methode ist ein weiteres Verfahren, das vor allem in der Spieleindustrie eingesetzt wird, um „Sehen“ virtueller Spielfiguren zu simulieren [11]. Bei der Raytracing-Methode werden mit Hilfe sogenannter Strahlen (engl.: Rays) Objekte in einem Sichtfeld lokalisiert. Ein Strahl besteht dabei aus zwei Elementen: Richtungvektor und Ausgangspunkt. Der Ausgangspunkt des Strahls ist zugleich der Standort des virtuellen Agent in der Szene. Hat ein Strahl Schnittpunkte mit einem Objekt in der Szene, gilt es als sichtbar. Schneidet der Strahl mehrere Objekte gleichzeitig, gilt nur jenes Objekt als sichtbar, dessen Distanz zum virtuellen Menschmodell, am geringsten ist. Das Sichtfeld des Agenten entsteht dadurch, dass mehrere verschieden ausgerichtete Strahlen gleichzeitig eingesetzt werden, siehe hierzu Abbildung 2. [11, 14, 3]

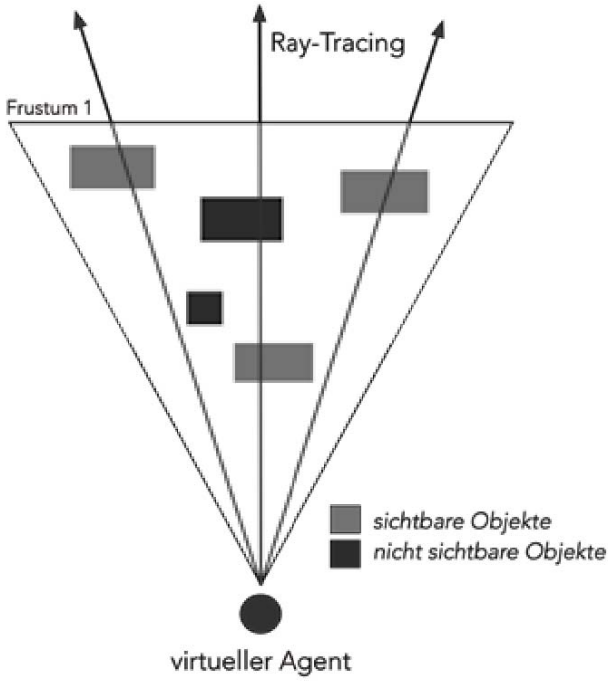

Abbildung 2: Ray-Tracing

3 Umsetzung

Für die Umsetzung der Wahrnehmung des virtuellen Agenten, wird ein Sensor umgesetzt, der wie in [4, 2] die geometrischen Informationen der dreidimensionalen Szene auswertet. Dabei identifiziert der Sensor Objekte in einem Sichtfeld abhängig der Position des virtuellen Menschmodells und dessen Ausrichtung. Wurden die Objekte beziehungsweise deren eindeutige Kennung vom Sensor identifiziert, liest dieser anhand der Kennungen die semantischen Informationen der Objekte aus einer Datenbank aus und leitet diese Informationen an den Softwareagenten weiter.

Die nachfolgend vorgestellten Verfahren verwenden nicht die Geometrie der Objekte in der Szene, sie benutzen stattdessen deren Hüllkörper beziehungsweise Boundingboxen, um mit Hilfe derer geringeren geometrischen Komplexität die Verfahren, zu beschleunigen. Die verwendeten Boundingboxen sind achsenorientiert und werden durch zwei Punkte definiert, einem *n*- und *p*-Vertex. Der *n*-Vertex ist dabei der Eckpunkt der Boundingbox, der die niedrigsten Koordinatenwerte aufweist. Der *p*-Vertex beschreibt den Punkt mit den größten Koordinatenwerten. [1]

3.1 View-Frustum Culling

Da das menschliche Verhalten bei der Wegfindung simuliert wird, muss der Informationszugriff über die Umgebung des virtuellen Agenten eingeschränkt werden. Dies wird damit begründet, da ein Mensch nur ein eingeschränktes Sichtfeld besitzt und somit beispielsweise Objekte die hinter ihm liegen, nicht wahrnehmen kann. Aus diesem Grund wird ein Sichtfeld wie in [2] simuliert, welches aus der Position und der Orientierung des virtuellen Menschmodells resultiert. Um dies zu realisieren, wird ein Verfahren aus der Computergrafik verwendet, das View-Frustum Culling [1]. Bei dieser Methode wird ein Sichtvolumen beziehungsweise Frustum erzeugt, mit dessen Hilfe Objekte die außerhalb des Sichtvolumens liegen, ausgeschlossen werden. Ein Frustum besteht aus sechs Flächen, die aus der Position und der Ausrichtung des virtuellen Menschmodells erzeugt werden. [1] Wie in Abbildung 3 zu sehen ist werden drei Vektoren *f*, *u* und *r* generiert. Der erste Vektor ist der Richtungsvektor *f* welcher die selbe Ausrichtung besitzt, wie das virtuelle Menschmodell *C*. Zusätzlich zu diesem, werden zwei Normalvektoren *r* und *u* erzeugt die orthogonal zum Richtungsvektor liegen. Anhand des Richtungsvektors und einer definierten Distanz *d* wird ein Punkt *F* errechnet. Ausgehend von diesem und der definierten Breite (*width*) und Höhe (*height*) der Flächen die orthogonal zum Richtungsvektor liegen, werden die Eckpunkte *P* des Sichtvolumens errechnet.

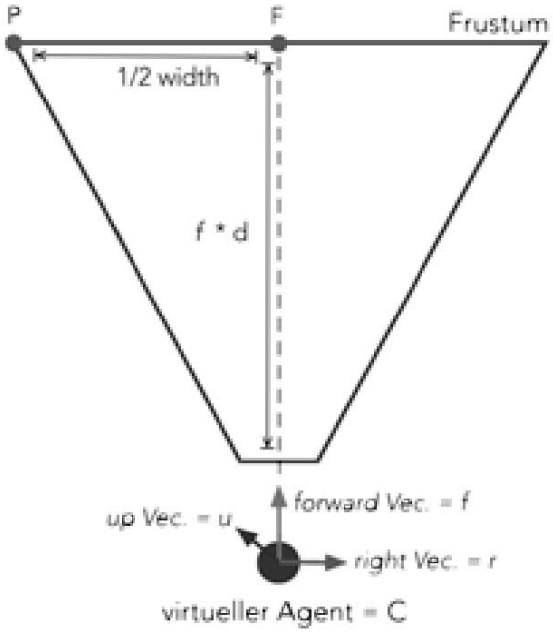

Abbildung 3: Ermittlung der Eckpunkte eines View-Frustums

$$F = C + f * d$$

$$P = F + (1/2 * height) * u - (1/2 * width) * r$$

Mithilfe zweier Eckpunkte *A* und *B* und dem Ausgangspunkt *D* wird im Anschluss die Normale *n* der Fläche aus *A*, *B* und *D* errechnet und anhand dieser getestet, ob ein Objekt innerhalb oder außerhalb des Sichtvolumens liegt.

$$n = \vec{DA} * \vec{DB}$$

Bei dem Test ob ein Objekt innerhalb des Sichtvolumens liegt, wird der orthogonale Abstand *d* der Eckpunkte *M* der Boundingbox des Objektes zu den sechs Flächen des Frustums errechnet. Zeigen die Normalen der Flächen des Sichtvolumens nach außen, liegt ein Punkt *M* außerhalb des Volumens, wenn dessen orthogonale Distanz *d* zu einer Fläche einen positiven Wert besitzt.

$$|n| = \sqrt{n^2}$$

$$d = \frac{n * M - 1}{|n|}$$

3.2 Ray-Tracing

Durch das View-Frustum Culling kann überprüft werden, ob ein Objekt innerhalb des Sichtfeldes liegt. Verdeckungen der Objekte untereinander werden hierbei nicht erkannt. Da nur „sichtbare“ Objekte für den virtuellen Agenten relevant sind, um ein möglichst menschliches Verhalten zu simulieren, müssen die vollständig verdeckten Objekte identifiziert werden. Ein Verfahren, das das Sichtbarkeitsproblem löst, ist das aus Kapitel 2.2 erwähnte Ray-Tracing, welches im Zuge dieser Arbeit für den Sichtsensor des Softwareagenten realisiert wurde.
Ein Strahl besteht aus einem Ausgangspunkt und einer Ausrichtung. Um zu untersuchen, ob eine Boundingbox von einem Strahl mit Ausrichtung d und Richtungsvektor o geschnitten wird, werden der n-Vertex und der p-Vertex ($p1$ und $p2$) der Boundingbox mit dem Strahl verrechnet, wodurch sich Schnittpunkte max und min ergeben:

$$min = \frac{p_1 - o}{d}$$

$$max = \frac{p_2 - o}{d}$$

Um ein Sichtfeld zu erzeugen, sind mehrere Strahlen mit verschiedenen Ausrichtungen notwendig. Hierfür werden maximale Höhe und Breite des Sichtfeldes definiert und die Anzahl der Strahlen in der Höhe und Breite festgelegt. Anhand dieser Angaben und der Position des virtuellen Menschmodells und dessen Ausrichtung, können die einzelnen Punkte für das Sichtfeld und die Ausrichtung der Strahlen errechnet werden, siehe Abbildung 4.

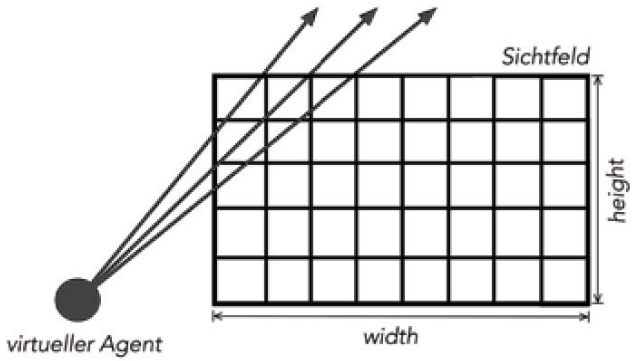

Abbildung 4: Erstellen eines Ray-Sichtfeldes

3.3 Occlusion-Culling

Das Rendern eines Bildes und das anschließende Auswerten dieses, gilt als zeitintensiv [vgl. 3, S. 11]. Da bei dem betrachteten Szenario mehrere Agenten gleichzeitig agieren und dabei ihre dreidimensionale Umwelt verzögerungsfrei wahrnehmen müssen, wurde entschieden den Occlusion-Culling Ansatz zu testen, um das Sichtbarkeitsproblem zu lösen [3, 1, 9]. Hierbei werden potenziell sichtbare Objekte (eng.: potentially visible objects, kurz PVO) mit Hilfe eines View-Frustums ermittelt. Anschließend müssen Occluder (dt.: Verdecker) identifiziert werden. Anhand dieser wird ein sogenanntes Occludee beziehungsweise Verdeckungsvolumen erzeugt, mit dessen Hilfe verdeckte PVOs, also Objekte die sich vollständig innerhalb des Verdeckungsvolumen befinden, als „sichtbar“ ausgeschlossen werden können. [3, 1, 9, 18] Siehe hierzu Abbildung 5. Um das Occlusion-Culling

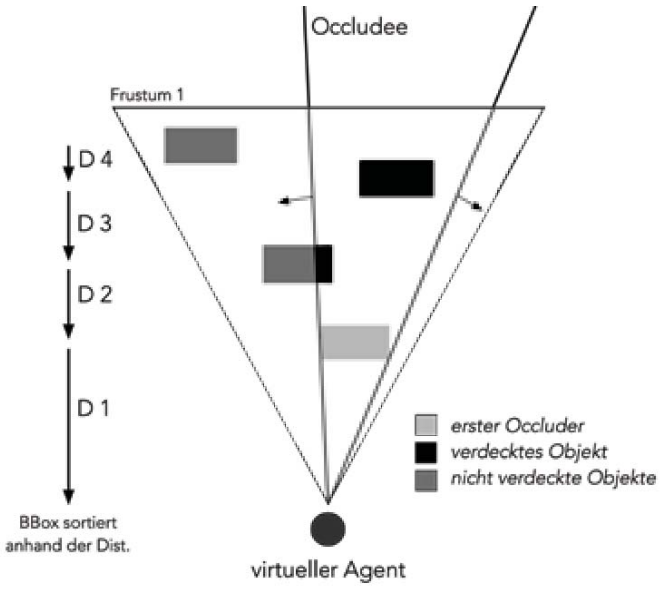

Abbildung 5: Ermittlung des erst zu betrachtenden Occluder und Erzeugung eines Verdeckungsvolumens

zu realisieren, werden mit dem in Kapitel 3.1 vorgestellten View-Frustum, die POVs ermittelt. Die Liste der PVOs wird anschließend

anhand derer Distanz zur Position des virtuellen Menschmodells hin sortiert. Im nächsten Schritt wird das PVO, welches den geringsten Abstand zum Menschmodell besitzt, aus der Liste ausgewählt. Das Verdeckungsvolumen wird anschließend aus den Extrempunkten der Boundingbox (BBox) des ausgewählten POVs aus Sicht des Menschmodells erstellt. Hierfür wurden die Boundingboxen um weitere Eckpunkte und einem Zentrum erweitert, wie in Abbildung 5 zu sehen ist. Für die Ermittlung der Extrempunkte, werden jeweils zwei Vektoren erstellt und anschließend deren Winkel zueinander berechnet. Für die Vektoren wird die Position des Menschmodells *D*, das Zentrum *C* und ein Eckpunkt *P* der Boundingbox verwendet. Der Vektor *DC* gilt als Vergleichsvektor, mit dessen Hilfe der Winkel zum Vektor *DP* errechnet wird. Wurden für alle Eckpunkte der Boundingbox diese Winkel ermittelt, werden je nach Orientierung des Objektes in Bezug zur Position des Menschmodells, vier beziehungsweise sechs Extrempunkte (Eckpunkte mit dem größten ermittelten Winkel) ausgewählt. Siehe hierzu Abbildung 6. Mithilfe dieser Ex-

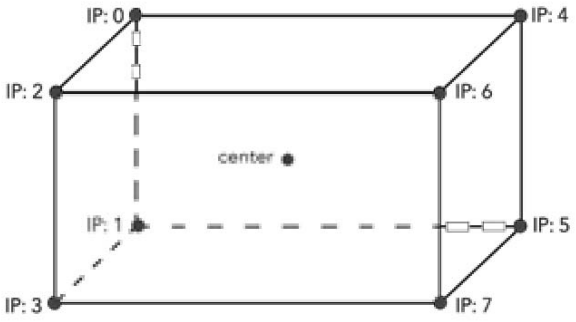

Abbildung 6: Erweiterte Boundingbox mit eindeutiger Kennung für Eckpunkte

trempunkte kann ein Frustum erstellt und alle übrigen PVOs auf eine Verdeckung getestet werden. Hierbei werden die einzelnen Eckpunkte der Boundingboxen dieser PVOs betrachtet und als verdeckt gekennzeichnet, falls diese innerhalb des Verdeckungsvolumens liegen. Ist ein Objekt vollständig verdeckt, sind also alle Eckpunkte der Boundingbox als verdeckt markiert, wird das betrachtete Objekt aus der Liste der PVOs entfernt. Wurden alle PVOs betrachtet, sind nur noch Objekte in der Liste der PVOs enthalten, die von der Position des virtuellen Menschmodells „sichtbar“ sind.

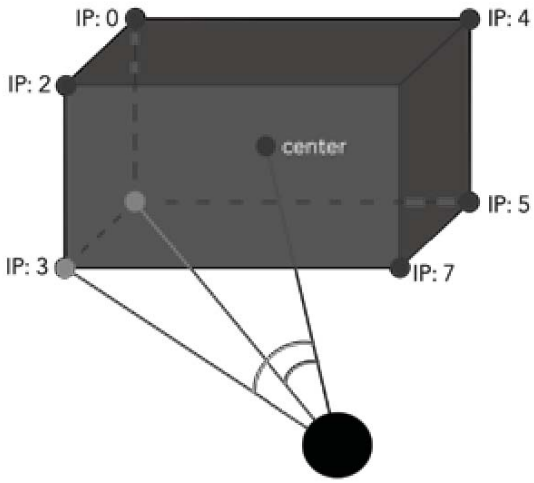

Abbildung 7: Ermittlung der Extrempunkte aus der Sicht des virtuellen Menschmodells. Der Punkt mit der IP: 3 gilt hier als Extrempunkt

4 Vergleich beider Verfahren

4.1 Laufzeitanalyse

Ray-Tracing Ansatz

Die Laufzeit des Ray-Tracing Ansatzes ist Abhängig von den erzeugten Strahlen und den zu testenden Objekten. In Abbildung 8 ist der Pseudocode des Ansatzes dargestellt.

Um einen Strahl zu erzeugen, ist die Position und die Ausrichtung des Strahls notwendig. Hierfür werden zwei *for*-Schleifen in den Zeilen 12 und 14 genutzt und anschließend die Ausrichtung des Strahls in Zeile 16 bis 20 errechnet. Diese Schleifen werden mit der Anzahl der definierten Zeilen und Reihen durchlaufen. In Zeile 22 wird der Funktion *findRayIntersection()* der zuvor definierte Strahl und die Liste der potenziell sichtbaren Objekte *n* aus dem ersten Frustumtest übergeben. Diese Funktion überprüft dabei, welche Objekte sich mit dem Strahl schneiden und gibt als Resultat das geschnittene Objekt zurück, das die geringste Distanz zum virtuellen Menschmodell aufweist. Der Algorithmus wird immer *columns* x *rows* x *n* - mal durch-

```
1
2
3   RAY-TRACING(agentOrigin, agentDirection, objectsInViewFrustum, width, height, rows, columns)
4
5   visibleObjects[] = 0
6
7   right = generateRightvector(agentDirection)
8   up = generateUpvector(agentDirection)
9
10  center = agentOrigin + (agentDirection * distance)
11
12  for i = 0 to columns
13
14      for j = 0 to rows
15
16          point = center + (up * (height/2 - ((height/columns) * i))) - (right * (width/2 - (width/rows) * j))
17
18          direction = point - origin
19
20          ray[direction, origin]
21
22          hitObject = findRayIntersection(objectsInViewFrustum, ray)
23
24          visibleObjects.push(hitObject)
25
26  return visibleObjects
```

Abbildung 8: Pseudocode: Ray-Tracing Algorithmus

laufen. *columns* und *rows* sind Konstante, wodurch die Laufzeit des Algorithmus von der Anzahl der zu prüfenden Objekte abhängt. Für den Ray-Tracing-Algorithmus gilt die Laufzeitkomplexität O(n).

Occlusion-Culling Ansatz

Der Occlusion-Culling Ansatz setzt voraus, dass die potenziell sichtbaren Objekte im Voraus nach deren Distanz zum Menschmodell sortiert sind. Der Sortieralgorithmus wird in dieser Ausarbeitung nicht näher betrachtet, für diesen gilt für *n* zu sortierende Objekte das Laufzeitverhalten O(n(n-1)/2). Der Algorithmus, der die sortierten Objekte auf Verdeckungen überprüft, ist als Pseudocode in Abbildung 9 dargestellt.

In Zeile 7 wird eine while-Schleife maximal so oft durchlaufen, bis alle potenziell sichtbaren Objekte *n* betrachtet wurden. Hierfür wird bei jedem Durchgang, siehe Zeile 5 beziehungsweise 30, ein Objekt, beziehungsweise dessen Boundingbox aus der sortierten Liste entfernt und mit diesem in Zeile 9 ein neues Verdeckungsvolumen generiert. Ab Zeile 11 werden anschließend alle in der Liste verbliebenen Objekte darauf getestet, ob diese vollständig im Verdeckungsvolumen liegen. Hierfür werden zwei *for*-Schleifen benötigt. Die Schleife in Zeile 13 wird *j*-mal durchlaufen, wobei für *j* die Anzahl der zu prüfenden Extrempunkte des zu testenden Objektes gilt. Die *for*-Schleife in Zeile 16 wird *k*-mal durchlaufen, wobei für *k* die Anzahl der Flächen des Verdeckungsvolumen gilt. Ab Zeile 17 wird überprüft, ob der ausgewählte Extrempunkt innerhalb des Verdeckungsvolumens liegt. Ist dies der Fall, wird dieser als „nicht sichtbar“ gekennzeichnet. Erst wenn alle Punkte als nicht sichtbar gekennzeichnet sind, wird das getestete Objekt in Zeile 27 aus der Liste der potenziell sichtbaren Objekte entfernt. Wenn kein Objekt ein anderes verdeckt, wird die *for*-Schleife in Zeile 11 maximal (n-1)/2 mal durchlaufen. Die Anzahl der Durchläufe der *for*-Schleifen in Zeile 13 und 16 sind konstant, da die Anzahl der zu prüfenden Extrempunkte der Boundingboxen und der generierten Flächen des Verdeckungsvolumen konstant sind. Für die Laufzeitanalyse sind somit nur die *while*-Schleife und die *for*-Schleife in Zeile 11 relevant, wodurch für die Laufzeitkomplexität O(n(n-1)/2) gilt.

4.2 Laufzeittest

Die in Kapitel 3 vorgestellten Verfahren wurden in JavaScript umgesetzt, da die virtuelle Welt in XML3D realisiert ist und mit JavaScript direkt auf das virtuelle Menschmodell, die Objekte und deren Boundingboxen in der Szene zugegriffen werden kann. Nachfolgend werden die Ergebnisse eines Laufzeittests der

```
OCCLUSION-CULLING(sortedObjectsInViewFrustum, agentOrigin)

visibleObjects[] = 0
objectTest = 0
nearestObject = sortedObjectsInViewFrustum.pop

while nearestObject != 0

    newFrustum = generateFrustum(nearestObject, agentOrigin)

    for i = 0 to sortedObjectsInViewFrustum.länge

        for j = 0 to sortedObjectsInViewFrustum[i].anzahlPunkte
            testPoint = sortedObjectsInViewFrustum[i].point[j]

            for k = 0 to newFrustum.anzahlFlächen
                distancePoint = getDistanceToPlane(newFrustum.plane[k], testPoint)

                if distancePoint > 0 then
                  testPoint.visibility = 1
                  objectTest = 1

                else
                  testPoint.visibility = 0

            if objectTest != 1
               sortedObjectsInViewFrustum.remove(sortedObjectsInViewFrustum[i])

    visibleObjects = visibleObjects + nearestObject
    nearestObject = sortedObjectsInViewFrustum.pop
```

Abbildung 9: Pseudocode: Occlusion-Culling Algorithmus

umgesetzten Verfahren vorgestellt. Der Laufzeittest wurde auf einem Windows 7 System mit einem Intel Core i7 bei einer Taktung von 2,80 GHz und 8 GB Arbeitsspeicher durchgeführt. Als Browser wurde Chrome in Version 33.0.1750.146 m verwendet. Bei einem Testlauf läuft der virtuelle Agent eine definierte Wegstrecke ab und untersucht dabei Objekte in der Szene auf deren Sichtbarkeit. Siehe hierzu Abbildung 9. Bei jedem Zeitintervall beziehungsweise bei jeder Positionsänderung des virtuellen Menschmodells wird das zu testende Verfahren eingesetzt. Bei einem Zeitintervall wird untersucht, wie viele Objekte als sichtbar markiert werden und wie viel Zeit dafür benötigt wird. Es wurden zwei Szenarien untersucht. Beim ersten Szenario waren 47 Objekte im Szenario enthalten. Diese wurden so platziert, dass es keine Verdeckungen gab, damit der Occlusion-Ansatz alle Objekte als Occluder ansieht. Im zweiten Szenario, wurden 310 Objekte in der Szene verwendet die 1/7 der Größe der Objekte im ersten Szenario entsprachen. Bei einem Testlauf wurden 12 x 60 Zeitintervalle untersucht. Für jedes Szenario wurden 12 Testläufe durchgeführt. Die Auswertung der Tests ist in Tabelle 1 und 2 im Appendix dargestellt. Werden die Ergebnisse der Laufzeittests betrachtet, so ist zu erkennen, dass der Laufzeitunterschied der Ansätze bei einer erhöhten Anzahl an Objekte erkennbar ist. Der Anstieg der Gesamtzeit entspricht jedoch nicht dem Laufzeitverhalten der Verfahren. So stieg die Gesamtzeit bei dem Ray-Tracing Ansatz nicht linear mit der Anzahl der Objekte an. Dies resultiert aus Konstanten die ebenfalls in die Zeitermittlung einfließen. So werden immer 140 Strahlen unabhängig der Objektanzahl *n* erzeugt. Der Laufzeittest hat zudem gezeigt, dass die Genauigkeit und die Laufzeit von der Objektgröße abhängt. So wurden bei dem Ray-Tracing Ansatz 40 % der zu sehenden Objekte als sichtbar markiert, wenn diese um das siebenfache verkleinert wurden, die Anzahl der Strahlen jedoch konstant blieb. Siehe hierzu Abbildung 10 im Appendix. Dies resultiert aus der Anzahl und Verteilung der Strahlen im Sichtfeld. Die Größe eines Objektes beeinflusst auch den Occlusion-Culling Ansatz. Verdeckt beispielsweise das erste zu untersuchende Objekt alle anderen Objekte, werden diese aus der Liste der PVOs entfernt und somit nicht weiter betrachtet.

Abbildung 10: Szenario: Laufweg (Linie) des virtuellen Agenten und dessen gesehene Objekte (hell)

5 Fazit und Ausblick

Der Vergleich beider Verfahren hat gezeigt, dass das Ray-Tracing Verfahren prinzipiell schneller als das Occlusion-Culling ist, dessen Schnelligkeit beziehungsweise Genauigkeit von der Größe der Objekte in der Szene abhängt. Eine genauere Beurteilung der Verfahren, kann erst vollzogen werden, wenn die dreidimensionale Umgebung für das Szenario erstellt wurde. Denkbar ist der Einsatz beider Verfahren in verschiedenen Situationen. Der Ray-Tracing Ansatz kann verwendet werden, wenn der virtuelle Agent lediglich große Objekte, wie beispielsweise Regale oder markante Punkte im Raum wahrnehmen muss, um sich im Raum zu orientieren. Der Occlusion-Culling Ansatz, kann eingesetzt werden, wenn der virtuelle Einkäufer zum Beispiel kleine Objekte in einem Regal Wahrnehmen muss, da er eine bestimmte Ware sucht. Um beide Verfahren in deren Laufzeit zu optimieren, kann der Ansatz der hierarchischen Unterteilung des Raumes wie in [4, 2] verfolgt werden. Hierdurch müssen nur die Objekte auf deren Sichtbarkeit getestet werden, wenn deren Elternobjekt als sichtbar gelten. Weitere Ansätze zur Einteilung des Raums, sind das BSP-, das Octree- oder das K-D-Baum-Verfahren [16, 13, 17].

Literatur

[1] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5:9–22, 2000.

[2] Christophe Bordeux, Ronan Boulic, and Daniel Thalmann. An efficient and flexible perception pipeline for autonomous agents, 1999.

[3] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER*, 9, 2003.

[4] Nathalie Farenc, Ronan Boulic, and Daniel Thalmann. An informed environment dedicated to the simulation of virtual humans in urban context. In *PROCEEDINGS OF EUROGRAPHICS'99*, pages 309–318, 1999.

[5] Martin Griss, Edited George, T. Heineman, Ph. D, Ph. D, and William Councill. Software agents as next generation software components. In *In "Component-Based Software Engineering," Edited by*, pages 641–657. Addison-Wesley, 2001.

[6] Hansrudi Noser and Daniel Thalmann. Synthetic vision and audition for digital actors. In *Proc. Eurographics '95*, pages 325–336, 1995.

[7] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11:205–244, 1996.

[8] Sejin Oh, Woonhyuk Baek, and Woontack Woo. Synthetic vision-based perceptual attention for augmented reality agents, 2010.

[9] Ioannis Pantazopoulos and Spyros Tzafestas. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems*, 35:123–156, 2002.

[10] Christopher Peters. A perceptually-based theory of mind for agent interaction initiation., 2006.

[11] Christopher Peters, Ginvera Castellano, Matthias Rehm, Elisabeth Andre©, Amaryllis Raouzaiou, Kostas Rapantzikos, Kostas Karpouzis, Gaultiero Volpe, Antonio Camurri, and Asimina Vasalou. Fundamentals of agent perception and attention modelling. In *Emotion-Oriented Systems*, pages 293–319. Springer Berlin Heidelberg, 2011.

[12] Christopher Peters and Carol O'Sullivan. Synthetic vision and memory for autonomous virtual humans, 2002.

[13] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics*, 13:445–460, 1989.

[14] T. Steel, Dane Kuiper, and Rym Z. Wenkstern. Virtual agent perception in multi-agent based simulation systems. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 2, 2010.

[15] Daniel Thalmann, Nathalie Farenc, and Ronan Boulic. Virtual human life simulation and database: why and how. pages 471–479, 1999.

[16] Csaba D. Tóth. Binary space partitions: Recent developments, 2005.

[17] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[18] Hansong Zhang. Effective occlusion culling for the interactive display of arbitrary models, 1998.

[19] Ingo Zinnikus, Xiaoqi Cao, Matthias Klusch, Christopher Krauss, Andreas Nonnengart, Torsten Spieldenner, and Philipp Slusallek. A collaborative virtual workspace for factory configuration and evaluation, 2013.

APPENDIX

Tabelle 1: Testszenario 1: 47 Objekte in der Szene ohne Verdeckungen

	Zeit in ms	Größe Objekt	Identifikation sichtbarer Objekte	n Objekte	Tests pro n	Komlpexität	Tests gesamt pro Zeitintervall
View-Frustum	Ø 0,2485	1/1		47	1	O(n)	47
Ray-Tracing	Ø 1,637	1/1	100 %	ca. 8	140	O(n)	1120
Occlusion-Culling	Ø 0,656	1/1	100 %	ca. 8	48	O((n(n-1)/2)	1344

Tabelle 2: Testszenario 1: 310 Objekte bei 1/7 Größe ohne Verdeckungen

	Zeit in ms	Größe Objekt	Identifikation sichtbarer Objekte	n Objekte	Tests pro n	Komlpexität	Tests gesamt pro Zeitintervall
View-Frustum	Ø 0,3235	1/7		310	1	O(n)	47
Ray-Tracing	Ø 1,957	1/7	ca. 40%	ca. 43	140	O(n)	1120
Ray-Tracing	Ø 19,524	1/7	ca. 100%	ca. 43	2125	O(n)	91375
Occlusion-Culling	Ø 2,850	1/7	ca. 100%	ca. 43	48	O((n(n-1)/2)	43344

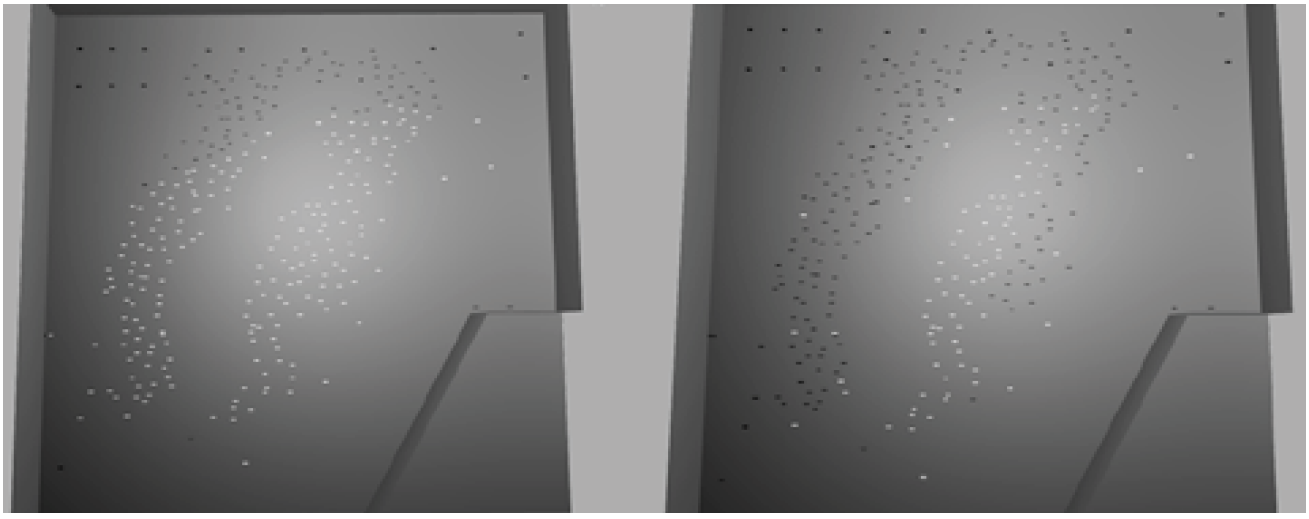

Abbildung 11: Genauigkeitsvergleich: 310 Objekte bei 1/7 Größe. Occlusion-Culling (links), Ray-Tracing mit 140 Strahlen (rechts)